### Some Dynamic Analyses for Concurrency

<table>
<thead>
<tr>
<th>Data race</th>
<th>Happens Before</th>
<th>Eraser</th>
<th>DJIT+</th>
<th>Goldilocks</th>
<th>FastTrack</th>
</tr>
</thead>
<tbody>
<tr>
<td>Atomicity</td>
<td>Block-Based</td>
<td>Commit-Node</td>
<td>Velodrome</td>
<td>SideTrack</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Atomizer</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Deterministic Parallelism</td>
<td>SingleTrack</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Deadlock</td>
<td>GoodLock</td>
<td>Pulse</td>
<td>Agarwal06</td>
<td>Deadlock Fuzzer</td>
<td></td>
</tr>
</tbody>
</table>

Monday, July 20, 2009
RoadRunner

Instrumented bytecode

RoadRunner Instrumenter

Java bytecode

JVM

events

T1: begin_atomic
T2: acquire(lock3)
T2: read(x,5)
T1: write(y,3)
T1: end_atomic
T2: release(lock3)

Atomicity violations

Monday, July 20, 2009
Thread 1

...  
x = 3;  
...

Instrumented Thread 1

...  
preWrite(x)  
x = 3;  
postWrite(x)  
...

MyTool.java

void preWrite(x){
  ...
}

void postWrite(x){
  ...
}
Happens-Before (HB) Race

Thread 1

write x

Thread 2

write x

```java
Thread t1 = new Thread(){
    void run()
    {
        x = 1;
    }
};

Thread t2 = new Thread(){
    void run()
    {
        x = 2;
    }
};

static int x;

public static void main(){
    t1.start();
    t2.start();
}
```
**HB Race**

```
Thread 1
sync m { }
write x
sync m { }
Thread 2
```

**No HB Race**

```
Thread 1
sync m { }
write x
sync m { }
Thread 2
sync m { }
write x
sync m { }
```
What about adding yields?

HB Race

Thread 1

```java
sync m { }
write x
yield()
```

Thread 2

```java
sync m { }
write x
sync m { }
```

- ad hoc
- no guarantees
- can be complex
- difficult to maintain

... So how do we write traces to test tools?
Thinking up Test Cases

T1
begin
read x
read x
x = false
T2
TIDDLE: a domain specific language for describing traces

\[
\text{trace} ::= \text{op}^* \\
\text{op} ::= \text{rd} \ Tid \ Var \ (Val) \\
| \text{wr} \ Tid \ Var \ (Val) \\
| \text{acq} \ Tid \ Lock \\
| \text{rel} \ Tid \ Lock \\
| \text{fork} \ Tid \ Tid \\
| \text{join} \ Tid \ Tid \\
| \text{beg} \ Tid \ Label \\
| \text{end} \ Tid \ Label \\
\text{Tid} ::= \text{Int} \\
\text{Var} ::= \text{String} \\
\text{Val} ::= \text{Int} \\
\text{Label} ::= \text{String}
\]
quick way of testing
dynamic analysis

• Race condition:

```java
package feasibleTests;
import java.util.concurrent.BrokenBarrierException;
import java.util.concurrent.CyclicBarrier;

public class RaceCondition {
    static int x = 0;
    static CyclicBarrier cb = new CyclicBarrier(2);
    static CyclicBarrier cc = new CyclicBarrier(2);
    static int counter = 0;
    static int numThreads = 2;

    static public void await(CyclicBarrier c) throws BrokenBarrierException, InterruptedException {
        c.await();
    }

    static int getCounter() {
        return counter++;
    }

    public static void main(String[] args) {
        final Thread t2 = new Thread() {
            public void run() {
                try {
                    int _z = 0;
                    await(cc);
                    await(cb);
                    await(cc);
                    x = getCounter();
                    await(cb);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                } catch (BrokenBarrierException e) {
                    e.printStackTrace();
                }
            }
        };
        final Thread t1 = new Thread() {
            public void run() {
                try {
                    int _z = 0;
                    await(cc);
                    _z = x;
                    await(cb);
                    await(cc);
                    await(cb);
                } catch (InterruptedException e) {
                    e.printStackTrace();
                } catch (BrokenBarrierException e) {
                    e.printStackTrace();
                }
            }
        };
        t1.start();
        t2.start();
    }
}
```

acq 1 m
wr 1 x
rel 1 m
wr 2 x
public class SimpleRace {
    static int x = 0;
    static Object m = new Object();
    static CyclicBarrier cb = new CyclicBarrier(2);

    static public void await(CyclicBarrier c)
        throws BrokenBarrierException, InterruptedException {
        c.await();
    }

    public static void main(String[] args) {
        final Thread t1 = new Thread() {
            public void run() {
                ...
            }
        };
        final Thread t2 = new Thread() {
            public void run() {
                ...
            }
        };
        t1.start();
        t2.start();
    }
}
public void run() {
    try {
        synchronized(m) {
            await(cb);
            x = 1;
            await(cb);
        }
        await(cb);
        await(cb);
    } catch (...) { ... }
}

public void run() {
    try {
        await(cb);
        await(cb);
        await(cb);
        x = 0;
        await(cb);
    } catch (...) { ... }
}

Thread 1

1. acq m
2. wr x
3. rel m
4. wr x

public void run() {
    try {
        synchronized(m) {
            await(cb);
            x = 1;
            await(cb);
        }
        await(cb);
        await(cb);
    } catch (...) { ... }
}

public void run() {
    try {
        await(cb);
        await(cb);
        await(cb);
        x = 0;
        await(cb);
    } catch (...) { ... }
}
Unit Testing for Dynamic Analysis

- Easy to describe test case as a trace
- Compile trace to multithreaded test
  - barriers for determinism
  - avoid boilerplate and complexity
Some Dynamic Analyses for Concurrency

- Data race
  - Happens Before
  - Eraser
  - DJIT+
  - Goldilocks
  - FastTrack

- Atomicity
  - Block-Based
  - Atomizer
  - Commit-Node
  - Velodrome
  - Atom Fuzzer
  - SideTrack

- Deterministic Parallelism
  - SingleTrack
  - Burnim09

- Deadlock
  - GoodLock
  - Pulse
  - Agarwal06
  - Deadlock Fuzzer
Atomicity

The effect of an atomic code block can be considered in isolation from the rest of a running program.

- enables sequential reasoning
- atomicity violations often represent synchronization errors
- most methods are atomic
Thread 1

```
atomic {
  synchronized(n) {
    tmp = bal;
  }
  synchronized(n) {
    bal = tmp + 1;
  }
}
```

Thread 2

```
synchronized(m) {
  newVar = 0;
}
```

Serial Trace: Each atomic block executes contiguously
Thread 1

```java
atomic{
    synchronized(n){
        tmp = bal;
    }
    synchronized(n){
        bal = tmp + 1;
    }
}
```

Thread 2

```java
synchronized(m) {
    newVar = 0;
}
```

Thread 1

```
begin
acquire(n)
t1 = bal
release(n)
end
```

Thread 2

```
acquire(m)
newVar = 0
release(m)
```

Atomicity = Serializability
Thread 1

```
atomic{
    synchronized(n){
        tmp = bal;
    }
    synchronized(n){
        bal = tmp + 1;
    }
}
```

Thread 2

```
synchronized(n) {
    bal = 0;
}
```
Thread 1

atomic{
    synchronized(n){
        tmp = bal;
    }
    synchronized(n){
        bal = tmp + 1;
    }
}

Thread 2

synchronized(n){
    bal = 0;
}

Thread 1

begin
acquire(n)
t1 = bal
release(n)
end

Thread 2

acquire(n)
bal = 0
release(n)
• HB Atomicity Violation:

- `beg 1 b`
- `wr 1 x`
- `wr 2 x`
- `rd 1 x`
- `end 1 b`
QUICK WAY OF TESTING DYNAMIC ANALYSIS

- HB Atomicity Violation:
  
  ```
  beg 1 b
  acq 1 m
  rel 1 m
  acq 2 m
  rel 2 m
  acq 1 m
  rel 1 m
  end 1 b
  ```
QUICK WAY OF TESTING DYNAMIC ANALYSIS

- Serial trace
- Feasibly non-serializable

beg 1 b
acq 1 m
rel 1 m
acq 1 m
rel 1 m
end 1 b
acq 2 m
rel 2 m
Tiddle Compiler

Lexer (alex) → Parser (happy) → Translation to Java AST → PrettyPrinters (HughesPJ)

rd 1 x
wr 2 y
rd 1 y

class Foo {
    ...
}

Monday, July 20, 2009
Equivalent Traces

• Equivalent modulo happens-before
• Different total orderings of same HB partial ordering
• Analyses should behave same way
• Regression testing for analyses
Next Steps

• Support for other Java features
  • volatiles, wait / notify, etc.
• Bug capture
• Specification language
Not a Panacea

• Helps you with the known unknowns

• This won’t help you find the bugs you’re not expecting

• Doesn’t help you with the unknown unknowns
Conclusion

• Domain-specific language
  • describe bugs succinctly
• Generate Java test cases
  • multithreaded, deterministic
• Helps dynamic analysis development
  • 100+ Tiddle traces to date